Finding Nearest Larger Neighbors
A Case Study in Algorithm Design and Analysis

Tetsuo Asano\(^1\), Sergey Bereg\(^2\), and David Kirkpatrick\(^3\)

\(^1\) School of Information Science, JAIST, Japan
\(^2\) Department of Computer Science, University of Texas at Dallas, USA
\(^3\) Department of Computer Science, University of British Columbia, Canada

Abstract. Designing and analysing efficient algorithms is important in practical applications, but it is also fun and frequently instructive, even for simple problems with no immediate applications. In this self-contained paper we try to convey some of the fun of algorithm design and analysis. Hopefully, the reader will find the discussion instructive as well.

We focus our attention on a single problem that we call the All Nearest Larger Neighbors Problem. Part of the fun in designing algorithms for this problem is the rich variety of algorithms that arise under slightly different optimization criteria. We also illustrate several important analytic techniques, including amortization, and correctness arguments using non-trivial loop invariants.

We hope, in this modest way, to reflect our deep admiration for the many contributions of Kurt Mehlhorn to the theory, practice and appreciation of algorithm design and analysis.

1 What Is the ANLN Problem?

Here is a general definition of our All Nearest Larger Neighbors (ANLN) Problem: Given a set \(S\) of \(n\) objects, find, for each object \(x\) in \(S\), an object \(y\) in \(S\) (if one exists) that is (i) larger than \(x\) and (ii) at least as close to \(x\) as any other object \(z\) that is larger than \(x\). We implicitly assume that any two objects are comparable, that is, one of them is larger than the other or they are equal. As we shall see, the problem is simplified considerably if we can assume that all elements are distinct.

Although the ANLN problem seems very natural and worthy of study even without specific applications, it is easy to imagine scenarios in which it could arise. For example, in emergency situations we often rely on protocols for sending messages from all individuals to some specified leader (or the reverse). It is easy to see the desirability of a tree-like protocol where (i) individual links are “short” and (ii) nodes closer to the root (leader) have greater authority (measured,
perhaps, by transmission power/capacity). To enact such a protocol, where each node has an assigned authority, it suffices to associate with every node its closest neighbor with higher authority.

2 The ANLN Problem for Linear Arrays

We begin with the simplest possible situation: the objects are \( n \) real numbers presented in an array \( A[1..n] \). For each element \( A[i] \), we want to determine an element \( A[j] \) among those with keys larger than \( A[i] \) that is closest to \( A[i] \), that is for which \( |j - i| \) is minimized, with ties broken in favor of the lower indexed neighbor.

Of course, an element with the largest key has no such larger neighbor. For this reason, as well as to simplify the presentation of some of our algorithms, we will assume that the array \( A \) has been implicitly extended to \( A[-n..3n] \), with \( A[-n] = A[3n] = \infty \) and \( A[j] = -\infty \) for \( j \in [-n+1..0] \cup [n+1..3n-1] \). It should be clear that with this extension the nearest larger neighbors of all non-maximal elements of \( A[1..n] \) are unaltered and the nearest larger neighbor of all maximal elements of \( A[1..n] \) is \( A[-n] \).

[ANLN Problem for a Linear Array]

**Input:** An array \( A[1..n] \) of \( n \) real numbers.

**Output:** An array \( NLN[1..n] \) such that \( A[NLN[i]] \) is the nearest larger neighbor of \( A[i] \). If \( A[i] \) is a largest element in \( A[1..n] \) then \( NLN[i] = -n \).

Figure 1 shows an example of an array \( A \) containing 10 numbers together with its associated \( NLN \) array.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>87</td>
<td>32</td>
<td>12</td>
<td>54</td>
<td>28</td>
<td>35</td>
<td>14</td>
<td>61</td>
<td>18</td>
<td>53</td>
</tr>
<tr>
<td>( NLN )</td>
<td>-10</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 1. An example of an array \( A \) containing 10 elements together with its associated \( NLN \) array

2.1 A Simple Linear-Time Stack-Based Algorithm

The ANLN problem has a very straightforward linear-time solution by computing NLN values to both the left (LNLN) and right (RNLN), using a stack:
Algorithm 1. (Double Stack-Scan): Compute both Left and Right NLN values

Output: Associated array $NLN[1..n]$.

begin
1. initialize stack $S$ as $\emptyset$
2. for $i = 1$ to $n$ do
3. while $S \neq \emptyset$ and $A[i] \geq A[\text{top}(S)]$ do
4. pop($S$)
5. if $S = \emptyset$ then $LNLN[i] = -n$ else $LNLN[i] = \text{top}(S)$
6. push $A[i]$ onto $S$
7. reinitialize stack $S$ as $\emptyset$
8. for $i = n$ to $1$ do
9. while $S \neq \emptyset$ and $A[i] \geq A[\text{top}(S)]$ do
10. pop($S$)
11. if $S = \emptyset$ then $RNLN[i] = 3n$ else $RNLN[i] = \text{top}(S)$
12. push $A[i]$ onto $S$
13. for $i = 1$ to $n$ do
14. if $i - LNLN[i] \leq RNLN[i] - i$ then $LN[i] = LNLN[i]$ else $LN[i] = RNLN[i]$  
16. end

The analysis of Algorithm 1 is completely straightforward. While its linear time complexity is clearly (asymptotically) optimal, it comes at the cost of using linear space, both for the stack $S$ and the intermediate NLN results. In recognition of the constraints of software embedded in highly functional devices like digital cameras and scanners, there has been a considerable amount of attention paid recently to the development of algorithms that use only a modest amount of working space. Thus, we are motivated to ask to what extent this is possible for the ANLN problem.

In practice, it is quite reasonable to allow working space of size that is logarithmic in the input size, since otherwise it is quite hard to incorporate recursion in designing algorithms. In this paper we restrict our attention to even more severely space-restricted algorithms. An algorithm is called a constant-working-space algorithm if it satisfies the following conditions.

Input: Input data are provided using a read-only array.
Output: Output data must be written to a write-only array.
Recursion: If recursive calls are included in the algorithm, stack area for recursive calls is considered as a part of the working space.
Working space: Arrays may be available in the algorithm, but their sizes must be constant independent of input sizes. Each element (variable or array element) may contain $O(\log n)$ bits where $n$ is the input size.
2.2 A Simple Linear-Time Stack-Based Algorithm

The most obvious way to compute NLN values with limited space is to do so one at a time. Since we need to identify only the closer of the two (left and right) nearest larger neighbors, it is natural to adopt a bidirectional search strategy (extending the neighborhood outward one step at a time) until we find a larger element.

It is easy to see that the following algorithm satisfies the constant working-space conditions:

**Algorithm 2.** (Bidirectional Scan): Scan neighborhoods by increasing distance.

Output: Associated array $NLN[1..n]$.

begin
  for $i = 1$ to $n$ do
    $NLN[i] = -n$
    for $k = 1$ to $\max\{i - 1, n - i\}$ do
      if $i - k \geq 0$ and $A[i - k] > A[i]$ then
        $NLN[i] = i - k$
        break
      if $i + k \leq n$ and $A[i + k] > A[i]$ then
        $NLN[i] = i + k$
        break
  end
end

**Lemma 1.** Algorithm 2 correctly computes the array $NLN[1..n]$ in $O(n^2)$ time. Furthermore, there is an example which requires $\Omega(n^2)$ time.

**Proof.** The algorithm consists of double loops, and clearly runs in $O(n^2)$ time. It is also easy to see that if $n$ given numbers are all the same then Algorithm 1 takes $\Omega(n^2)$ time. $\square$

Somewhat surprisingly, Algorithm 2 is considerably more efficient, in the worst case, if we restrict attention to input arrays with no duplicate keys:

**Theorem 2. [Power of Bidirectional Search]**

Assuming that the elements of $A[1..n]$ are all distinct, Algorithm 2 computes the array $NLN[1..n]$ in $\Theta(n \log n)$ time in the worst case.

**Proof.** By the assumption that every input number is distinct, there is a unique largest element in $A[1..n]$. Let $d_i = |i - NLN[i]|$ and define $\sigma$ to be a permutation for which

$$d_{\sigma(1)} \geq d_{\sigma(2)} \geq \cdots \geq d_{\sigma(n-1)}. \tag{1}$$
Let \( k \in \{1..n\} \) and consider the distribution of the \( k \) elements \( A[\sigma(1)], \ldots, A[\sigma(k)] \) in the array (see Figure 2). The smallest distance between two such elements, realized say by \( A[\sigma(i)] \) and \( A[\sigma(j)] \), must be no more than \( n/(k+1) \).

By our element distinctness assumption the two elements \( A[\sigma(i)] \) and \( A[\sigma(j)] \) cannot be equal; without loss of generality assume that \( A[\sigma(i)] < A[\sigma(j)] \). Then \( d_{\sigma(k)} \leq d_{\sigma(i)} \leq n/(k+1) \).

Since the running time of the algorithm is \( \Theta(\sum_{k=1}^{n} d_k) \), the upper bound on the cost of Algorithm 2 follows from the fact that

\[
\sum_{k=1}^{n} d_k = \sum_{k=1}^{n} d_{\sigma(k)} \leq \sum_{k=1}^{n} \frac{n}{k+1} = O(n \log n).
\]  

(2)

It is also easy to show that there is an example for which Algorithm 2 takes \( \Omega(n \log n) \) time. Let \( n = 2^{t+1} - 1 \). We construct a permutation of \( (1, 2, \ldots, 2^{t+1} - 1) \) by (i) assigning the keys \( (1, 2, \ldots, 2^{t+1} - 1) \) to the nodes of a perfectly balanced binary tree \( T \) (of height \( t \)) in max-heap order, and (ii) listing the keys by an in-order traversal of \( T \). It is easy to see that all \( 2^{t-h} \) keys at height \( h < t \) have \( NLN \) distances equal to \( 2^h \). Thus the cost of Algorithm 2 is \( \Theta(t2^t) = \Theta(n \log n) \) in this case.

It remains to see if we can improve upon the simple bidirectional search strategy when the input array contains duplicate elements.

### 2.3 Improvements with Duplicate Keys

#### Two Distinct Numbers

Suppose that the input sequence contains elements in \( \{0, 1\} \).

**Lemma 3.** \( NLN \)-values for any \( \{0, 1\} \)-array \( A[1..n] \) can be found in \( O(n) \) time using constant working space.

**Proof.** First, set \( NLN[i] = 0 \) for all \( i \) such that \( A[i] = 1 \) since all such elements have no larger neighbor. It remains to show how to assign \( NLN[i] \) for all \( i \) such that \( A[i] = 0 \).

Suppose that at least one such element exists (otherwise, nothing remains to be done). Let \( A[i..j] \) be any maximal interval of 0 elements within \( A[1..n] \). If
\[ i = 1 \text{ then, by maximality, } A[j + 1] = 1 \text{ and so } NLN[x] = j + 1, \text{ for all } x \in [i, j]. \]

Similarly, if \( j = n \) then \( NLN[x] = i - 1 \), for all \( x \in [i, j] \). Alternatively, it must be the case that both \( A[i - 1] = 1 \) and \( A[j + 1] = 1 \) and hence \( NLN[x] = i - 1 \), for \( x \in [i, m] \), and \( NLN[x] = j + 1 \), for \( x \in [m + 1, j] \), where \( m = \lceil (i + j)/2 \rceil \).

Since both identifying maximal sequences of 0’s and assigning \( NLN \)-values, as specified, can be done with constant working space, the result follows. \( \square \)

**Remark.** Suppose that the input sequence contains elements in \{−1, 0, 1\}. It is straightforward to modify the algorithm above to compute \( NLN[x] \) for all \( x \) such that \( A[x] = 0 \).

**k Distinct Numbers.** Suppose now that the input array \( A \) contains \( k \) distinct numbers. Assume for simplicity that they are 1, 2, \ldots, \( k \). By definition \( NLN[x] = 0 \), for all \( x \) such that \( A[x] = k \). For each \( v = 1, 2, \ldots, k - 1 \), we use the above algorithm using the following implicit transformation of \( A \). Replace \( A[j] \) by -1 (respectively, 0 or 1) if it is \(< \) (respectively, \( = \) or \( > \) \( v \)). By the result of the previous subsection, the running time to compute the \( NLN \)-values associated with the 0-elements (of the implicit array) is \( O(n) \) and the working space is \( O(1) \). Thus, the total running time is \( O(kn) \).

**General Algorithm–Not Necessarily Distinct Inputs.** Suppose that the input contains \( k \) distinct numbers, where \( k \) is any number from 1 to \( n \). We design an algorithm that achieves the running time of two above algorithms: the time is \( O(n \min\{k, \log n\}) \).

Our algorithm is presented in pseudo code as Algorithm 3. We assume that the input is given by an array \( A[1..n] \) of real numbers. Recall that, to make \( NLN[i] \) defined for largest elements in \( A[1..n] \) and to avoid cluttering the pseudo code with indexing checks, we have further assumed that the array \( A \) has been extended so that \( A[-n] = A[3n] = \infty \) and \( A[j] = -\infty \), for \( j \in [-n, 1.0] \cup [n + 1.3n] \).

The high level structure of Algorithm 3 is a repetition, for all elements \( a \in [1..n] \) of a process that we call ScanFrom(\( a \)). If, for some \( s > 0 \), the elements of the subarrays \( A[a - 2..a - s - 1] \) and \( A[a + 1..a + s] \) all have value at most \( A[a] \), the elements of the subarray \( A[a - s + 1..a - 1] \) all have value less than \( A[a] \), and the element \( A[a - s] \) has value equal to \( A[a] \), then we say that the index \( a \) is passive; otherwise, it is active. ScanFrom(\( a \)) either aborts without reporting any \( NLN \)-values (having discovered that index \( a \) is passive) or it reports the \( NLN \)-value of \( a \) and all indices \( i > a \) for which \( A[i] = A[a] \), up to (but not including) the first (smallest) such index that is active.

The algorithm is presented in a way that emphasizes the fact that the elements \( a \in [1..n] \) can be treated in any order (or even in parallel).

Output: Associated array $NLN[1..n]$.

begin
forall $a \in \{1, \ldots, n\}$ do
  /* ScanFrom($a$) */
  $x \leftarrow A[a]$;
  $s \leftarrow 1$; $e \leftarrow a$;
  while $A[a-s] < x$ and $A[a+s] \leq x$ do /* Invariant 1 */
    if $A[a+s] = x$ then $e \leftarrow a + s$;
    $s \leftarrow s + 1$;
  /* ($A[a-s] \geq x$ or $A[a+s] > x$) and Invariant 1 */
  if $A[a+s] > x$ then
    /* $A[r] > x$ */
    $r \leftarrow a + s$;
    /* $A[l] \leq r \Rightarrow r - a < a - l$ */
    else
      /* $A[a-s] \geq x$ and Invariant 1 */
      $p \leftarrow s$;
      while $A[a-p] \leq x$ do /* Invariant 2 */
        if $p = 2s$ then abort ScanFrom($a$) /* Index $a$ is passive */
        $p \leftarrow p + 1$;
        /* $A[a-p] > x$, $p \leq 2s$ and Invariant 2 */
        $l \leftarrow a - p$;
        /* $A[l] > x$ */
        $t \leftarrow a + s$;
        while $t - e < e - l$ and $A[t] \leq x$ do /* Invariant 3 */
          if $A[t] = x$ then $e \leftarrow t$;
          /* $A[t] \leq x \Rightarrow t - e \geq e - l$ */
          $t \leftarrow t + 1$;
        /* $A[l] \leq r \Rightarrow t - e \geq e - l$ */
        $r \leftarrow t$;
        /* Invariant 4 */
      if $r - a \geq a - l$ then $NLN[a] \leftarrow l$ else $NLN[a] \leftarrow r$;
      $j \leftarrow a$; $i \leftarrow a + 1$;
      while $i \leq e$ do /* Invariant 5 */
        if $A[i] = x$ then /* $j - l > i - j$ */
          if $r - i \leq i - j$ then /* Index $i$ is active */
            abort ScanFrom($a$);
          else /* Index $i$ is passive */
            if $r - i \geq i - l$ then $NLN[i] \leftarrow l$ else $NLN[i] \leftarrow r$;
          $j \leftarrow i$;
        $i \leftarrow i + 1$;
  end
The following invariants describe the evolving knowledge about the structure of the array $A$, and serve to support our proof of correctness.

**Invariant 1**

(i) $A[k] < x$, for $a - s < k < a$;
(ii) $A[a] = x$;
(iii) $A[k] \leq x$, for $a < k < e$;
(iv) $A[e] = x$; and
(v) $A[k] < x$, for $e < k < a + s$.

**Invariant 2**

(i) $A[k] \leq x$, for $a - p < k < a$;
(ii) $A[a - s] \geq x$;
(iii) $A[k] < x$, for $a - s < k < a$;
(iv) $A[a] = x$;
(v) $A[k] \leq x$, for $a < k < e$;
(vi) $A[e] = x$; and
(vii) $A[k] < x$, for $e < k < a + s$.

**Invariant 3**

(i) $A[k] \leq x$, for $l < k < a$;
(ii) $A[a] = x$;
(iii) $A[k] \leq x$, for $a < k < e$;
(iv) $A[e] = x$; and
(v) $A[k] < x$, for $e < k < t$.

**Invariant 4**

(i) $A[k] \leq x$, for $l < k < a$;
(ii) $A[a] = x$;
(iii) $A[k] \leq x$, for $a < k < e$;
(iv) $A[e] = x$;
(v) $A[k] < x$, for $e < k < r$;
(vi) $A[l] \leq x$;
(vii) $A[r] > x \land r - a < a - l$.

**Invariant 5**

(i) $A[k] \leq x$, for $l < k < j$;
(ii) $A[j] = x$;
(iii) $A[k] < x$, for $j < k < i$;
(iv) $A[k] \leq x$, for $i \leq k < e$;
(v) $A[e] = x$; and
(vi) $A[k] < x$, for $e < k < r$.

Confirmation of these invariant properties (at the appropriate locations in the algorithm) is a lengthy, but completely straightforward, exercise.

**Lemma 4.** All NLN-values assigned by Algorithm 3 are correct

**Proof.** NLN-values are assigned in lines 22 and 29. The correctness of these assignments follows immediately from Invariant 4. □

We refer to the interval of array indices encountered during ScanFrom($a$) as the *reach* of ScanFrom($a$). Index $u$ in the reach of ScanFrom($a$) satisfying $A[u] = A[a]$ is said to be a predecessor (respectively, successor) of index $a$ if $u < a$ (respectively, $u > a$). Note that if $a$ is passive then it must have a predecessor and hence, by transitivity, a nearest active predecessor.

**Lemma 5.** For all $a, 1 \leq a \leq n$, NLN[$a$] is assigned exactly once. If $a$ is active then NLN[$a$] is assigned during ScanFrom($a$). If $a$ is passive then NLN[$a$] is assigned during ScanFrom($u$), where $u$ is the nearest active predecessor of $a$.

**Proof.** It is straightforward to confirm that ScanFrom($a$) aborts at line 14 if and only if index $a$ is passive. Hence, if index $a$ is active then NLN[$a$] is reported at line 22. Since ScanFrom($a$) never reports NLN-values for indices $i < a$, or for active indices $i > a$, the lemma clearly holds for active indices.
Suppose now that index \( a \) is passive. Since ScanFrom(\( a \)) aborts at line 14, if \( NLN[\( a \)] \) is reported it must occur during ScanFrom(\( u \)), for one or more active indices \( u < a \). But, by line 26, \( NLN[\( a \)] \) is not reported during ScanFrom(\( u \)) if there exists an active index \( w \) between \( u \) and \( a \). Thus, if \( NLN[\( a \)] \) is reported it must occur during ScanFrom(\( u \)), where \( u \) is the nearest active predecessor of \( a \).

Suppose that \( a \) is the smallest passive index such that \( NLN[\( a \)] \) is not assigned during ScanFrom(\( u \)), where \( u \) is the nearest active predecessor of \( a \). By the minimality of \( a \), we can assume that all passive predecessors of \( a \) following \( u \) are assigned during ScanFrom(\( u \)). It follows, by lines 26-29, that \( a \) must not lie within the reach of ScanFrom(\( u \)). Hence, by Invariant 3, the gap separating \( a \) from its closest predecessor \( v \), must be at least the distance from \( v \) to the left reach of ScanFrom(\( u \)). But this contradicts our assumption that index \( a \) is passive.

One can easily convert \( A \) into an implicit array of distinct numbers by replacing \( A[i] \) by the pair \( \langle A[i], i \rangle \) and using lexicographic order on the pairs. Let \( d_i \) be the distance of \( A[i] \) to its nearest larger neighbor in this lexicographic order. Let

\[
D = \sum_{i=1}^{n} d_i.
\]

**Lemma 6.** Algorithm 3 runs in \( O(D) \) time and uses constant working space.

**Proof.** It is clear from inspection of the pseudo code that Algorithm 3 uses constant working space and that the cost associated with ScanFrom(\( a \)) is proportional to the size of the reach of ScanFrom(\( a \)). Thus, to show that Algorithm 3 uses \( O(D) \) time it suffices, in light of Lemma 6 to show that the reach of ScanFrom(\( a \)) is (i) \( O(d_a) \), if \( a \) is passive, and (ii) \( O(\sum_{i \in R(a)} d_i) \), where \( R(a) \) denotes the set of all indices whose \( NLN \)-values are set in ScanFrom(\( a \)), otherwise.

Case (i) is clear since, when ScanFrom(\( a \)) aborts, the reach of ScanFrom(\( a \)) is \([a - 2d_a, a + d_a]\). For case (ii), we first note that if the nearest larger neighbor of \( a \) has an index larger than \( a \), or if \( a \) has no successor within the reach of ScanFrom(\( a \)), then the reach of ScanFrom(\( a \)) is at most \([a - 2d_a, a + 2d_a]\). So we can assume that \( l \) (the left reach of ScanFrom(\( a \))) is the nearest larger neighbor of \( a \) and that \( a \) has one or more successors within the reach of ScanFrom(\( a \)). Let \( w \) denote the rightmost such successor.

By lines 18-20, the size of the reach of ScanFrom(\( a \)) is at most twice the gap from \( l \) to \( w \). Since the gap from \( l \) to \( a \) is at most \( 2d_a \), and the gap from any inactive successor \( u \) of \( a \) to its immediate predecessor is exactly \( d_u \), the result follows directly from Lemma 6 if all of the successors of \( a \) are inactive (and hence belong to \( R(a) \)). On the other hand, if \( a \) has an active successor, and \( v \) denotes the nearest such successor, then the gap between \( v \) and its immediate predecessor \( u \) must be less than the gap from \( l \) to \( u \) (otherwise \( v \) would not be in the reach of ScanFrom(\( a \))). Furthermore, since \( v \) is active, it also follows that the gap between \( v \) and the right reach of ScanFrom(\( a \)) is no more than the gap between \( v \) and \( u \). But, as we have already already argued, the gap between \( l \) and \( u \) is \( O(\sum_{i \in R(a)} d_i) \). It follows that the full reach of ScanFrom(\( a \)) is also \( O(\sum_{i \in R(a)} d_i) \). \( \square \)
Lemma 7. If array \( A[1..n] \) contains \( k \) distinct elements then \( D = O(n \min(k, \log n)) \). Furthermore, in the worst case, \( D = \Theta(n \min(k, \log n)) \).

Proof. The result when \( k > \log n \) follows immediately from the proof of Theorem 2. So suppose that \( k \leq \log n \). To show that \( D = O(nk) \) consider any number \( x \) in \( A[1..n] \). Let \( i_1, i_2, \ldots, i_r \) be the indices of \( x \), i.e. \( x = A[i_1] = A[i_2] = \ldots A[i_r] \). By definition, \( d_{i_p} \leq i_{p+1} - i_p \), for \( 1 \leq p < r \), and hence \( d_{i_1} + d_{i_2} + \ldots d_{i_r} = O(n) \). It follows that \( D = O(nk) \).

It remains to show that there exists a sequence of \( n \) numbers from the set \( \{0, 1, \ldots, k-1\} \) such that \( D = \Omega(nk) \). Suppose, for simplicity, that \( n = 2^{t+1} - 1 \). We construct a sequence by (i) assigning the key \( k-s \) to all \( 2^{s-1} \) nodes on level \( s \), for \( 1 \leq s < k \), of a perfectly balanced binary tree \( T \) (of height \( t \)) in maxheap order, (ii) assigning key \( 0 \) to nodes on all other levels and (iii) listing the keys by an inorder traversal of \( T \). It is easy to see that all \( 2^{k-x-1} \) keys of value \( x \) have \( NLN \) distances equal to \( 2^{r-k+x} \), for \( 1 \leq x < k-1 \). Thus the cost of Algorithm 3 is \( \Theta(2^k) = \Theta(nk) \) in this case. \( \Box \)

3 NLN Problem in Higher Dimensions

It is natural to consider the NLN problem in higher dimensions as well. In two dimensions, suppose we have an array \( A \) of size \( m \times m \). For each element \( A[i, j] \) we want to find a larger element \( A[p, q] \) that is nearest to \( A[i, j] \). We consider the situation where distance between two such array elements is measured using the \( L_1 \) metric \( \|p-i\| + |q-j| \) or the \( L_2 \) (Euclidean) metric \( \sqrt{(p-i)^2 + (q-j)^2} \).

A natural idea (that generalizes the one-dimensional bidirectional search) for solving the nearest larger neighbors problem is to examine neighbors in the increasing order of their distances. We call this algorithm the Distance Heuristic.

For \( L_1 \) distance the Distance Heuristic has a straightforward constant-space implementation that exploits the fact that the \( L_1 \) \( k \)-neighborhood of an element \( A[i, j] \) (the set of elements \( A[p, q] \) whose \( L_1 \) distance from \( A[i, j] \) is exactly \( k \)) forms a rhombus within \( A \) centered at \( A[i, j] \) and has a simple constant-space enumeration.

If the nearest larger neighbor of \( A[i, j] \) has \( L_1 \) distance \( d_{i,j} \), then the time required to determine \( NLN[i, j] \) is \( O(d_{i,j}^2) \). Let \( d_k \) denotes the \( k \)-th largest among the \( m^2 \) \( NLN \)-distances and consider the distribution of the \( k \) elements of \( A \) whose \( NLN \)-distances are at least \( d_k \). Since \( d_k \) is the smallest \( L_1 \) distance between two such elements, it must satisfy the inequality \( k(d_k/2)^2 \leq m^2 \). Thus we have \( d_k^2 \leq 4m^2/k \). It follows, by the same analysis used in the proof of Theorem 2, that the Distance Heuristic runs in \( O(m^4 \log m) \) time if all elements are distinct.

The problem of implementing the Distance Heuristic using \( L_2 \) distances is slightly more involved. The main difficulty is that there seems to be no efficient way of generating neighbors in increasing order of their \( L_2 \) distances from a fixed element \( A[i, j] \), using only constant working space. Fortunately, we do not really need to generate neighbors in strictly increasing order.
The idea is the following. We first find the $L_1$ NLN-distance $d_{i,j}$ associated with $A[i,j]$, as above. We then find the element, among all elements larger than $A[i,j]$ with $L_1$ distance between $d_{i,j}$ and $\sqrt{2}d_{i,j}$, that minimizes the $L_2$-distance from $A[i,j]$. (See the pseudocode for Algorithm 4 for more detail.)

**Algorithm 4. $L_2$-Distance Heuristic.**

**Input:** Array $A[1..m, 1..m]$ of keys

**Output:** Associated array $NLN[1..m, 1..m]$

begin
  for each element $A[i,j]$ in $A$ do
    // First Phase: find an $L_1$ nearest larger neighbor //
    $k = 0; D = \infty$
    repeat
      for each element $A[p,q]$ in the $L_1$ $k$-neighborhood of $A[i,j]$ do
          $D = (p-i)^2 + (q-j)^2$
          $NLN[i,j] = (p,q)$
        until $k = m$ or $D < \infty$
      if $D = \infty$ then $NLN[i,j] = (-m,-m)$ else
        $t = k$
      repeat
        for each element $A[p,q]$ in the $L_1$ $t$-neighborhood of $A[i,j]$ do
          if $A[p,q] > A[i,j]$ and $(p-i)^2 + (q-j)^2 < D$ then
            $D = (p-i)^2 + (q-j)^2$
            $NLN[i,j] = (p,q)$
          until $t > \sqrt{2}k$
    end
end

**Theorem 8.** The $L_2$-Distance Heuristic solves the Nearest Larger Neighbors Problem in $O(m^2 \log m)$ time for any $m \times m$ array $A$, under the assumption that all elements are distinct. Furthermore, there is an example of such an array for which $\Theta(m^2 \log m)$ time is required.

**Proof.** The correctness of the algorithm using the $L_1$ distance is obvious since we generate neighbors in the increasing order of the $L_1$ distances. For the $L_2$ distance we also generate neighbors using the $L_1$ distances. We exploit the fact that $L_1$ distance overestimates $L_2$ distance by a factor between $1$ and $\sqrt{2}$. See Figure 3.

The following example shows that the bound is tight, in the worst case. Suppose that $m = 2^k + 1$. We construct a $m \times m$ array $A$ with keys from $1$ to
\( m^2 = 2^{2k} + 2^{k+1} + 1 \) such that Distance Heuristic takes \( \Omega(m^2 \log m) \) time to solve NLN problem for \( A \). Our construction is recursive and assumes that the keys in the first and last row and column are the larger than any key in the interior of a given subarray. We initialize this invariant by assigning the \( 4m - 4 \) largest keys arbitrarily to the first and last rows and columns of \( A \). To continue we first assign the next largest key to the central position \( A[2^{k-1} + 1, 2^{k-1} + 1] \) and then the next \( 4m - 5 \) largest elements arbitrarily to the unassigned positions in the middle row and column of \( A \). In effect this partitions \( A \) into four subarrays of size \( 2^{k-1} \times 2^{k-1} \) (with overlapping boundaries) that satisfy the invariant assumption. Thus, we are free to continue the assignment of keys recursively in each of these submatrices.

The specified assignment has the property that that each of the \( 4t \) central elements assigned at the \( t \)-th level of recursion has a nearest larger neighbor at distance \( 2^{k-1-t} \), for \( 0 \leq t < k \). Thus, summing over these central elements only we see that the Distance Heuristic takes time at least \( \sum_{t=0}^{k-1} 4^t (2^{k-1-t})^2 \) which is \( \Theta(k2^k) \) or \( \Theta(m^2 \log m) \).

\[ \text{Fig. 3. The region to check whether a closer larger neighbor exists} \]

### 4 Extensions

It is not hard to see how to extend the Distance Heuristic (for both \( L_1 \) and \( L_2 \)) to work on \( d \)-dimensional arrays of total size \( m^d \) in time \( O(m^d \log m) \), assuming all keys are distinct. However, when \( d > 1 \) it remains an open problem how to achieve this same bound if the distinctness assumption is dropped.

For the one-dimensional ANLN problem, the reader may find it an interesting exercise to implement Algorithm 3 as a distributed (message passing) algorithm on a ring of processors.